e-speak

Contributed Services

Developer Release 3.01
June 2000

HEWLETT® PACKARD

Developer Release 3.01 June 2000

Contents

Chapter 1 Preface

This book describes the contributed applications of e-speak, a middleware product that simplifies creation of distributed electronic services.

E-speak provides the infrastructure needed to support the next level of internet business collaboration through development of e-services. That said, e-speak doesn't do anything terribly interesting by itself. The interest is in the applications that it can host rather than in the infrastructure.

This document describes the contributed services; applications written to show off what e-speak can do. From the virtual file-system that shows how refined a service can be by registering each file as a separate service/resource, to the minimalist tunnel service that shows how even applications written entirely without consideration of e-speak can still use e-speak, each sample application is written to take advantage of the features that the infrastructure provides.

Take a look at each and try them out. While these are unsupported applications, they show the core of what e-speak enables, and demonstrate how to write more complex applications using the e-speak APIs. And if you have suggestions (or code) that would improve them, please send them along so that they can be included in the next release.

Why Read This Book?

The goal of this book is to help you understand the contributed applications so that you can make best use of the ideas they represent. These example programs represent the best coding practices we have learned about our code, and our best guesses of how to make good use of the features it presents.

If you have a handle on the e-speak API, but are running into trouble when it comes to actually implementing a service, these applications give you more complex examples than those found in the documentation. In writing these applications, we tried to explore the following problems:

- What should be a service?
- How does a service restart reliably?
- How are services located and used?
- How can the system be used to manage transient services?
- Can objects be passed by reference; and can their stubs and skeletons be constructed on the fly?
- How to serve common interest groups?
- How should dynamic service location be used; and how does an application manage which resource to pick?

Scope of This Book

This book is divided into four chapters following this preface. Each chapter describes one of the contributed applications:

- Virtual File System
  A file-system implemented in e-speak that uses the core system to manage its files. Each file is registered as a separate service, and the core persistence is used to manage the file life-cycle instantiations of the service. The VFS includes a sample browser for Windows that is similar to the file system explorer included with the operating system.
- Print Service
  A tool for managing access to printers and locating the nearest or best printer for a job.

- Chat Service
  Instant messaging implemented as an e-speak service. The chat service allows discussion groups to form dynamically by allowing participants to express interest in a subject rather than having central management of a member list.

Within each chapter you can find a section describing what the application does and how it is architected, installation or initialization information, usage, how to configure or customize, and a full description of the application design.

Chapter 2 Virtual File System

VFS is a virtual file system that is built upon the e-speak infrastructure. VFS provides the ability to create and manage file objects (henceforth referred to as files) that are maintained somewhere in the e-speak environment. The files exist on some device in the Internet, however, just like with a web URL, the user doesn't really care what device hosts the file - he or she just wants to be able to access it. VFS files are stored hierarchically; each user has a workspace consisting of one or more cabinets. Each cabinet contains folders, which can hold other folders or individual files.

Where VFS extends beyond a typical file system is in its use of the unique features of e-speak. For example: the virtual folder hierarchy is unique to each user. Just as a web browser maintains its own list of bookmarks, each user is able to build their own hierarchy of files that are of interest to that user, and each user can refer to a file by what ever name is meaningful to them. They are not bound to a name that is common to all users.

VFS allows the user to create a workspace of cabinets that contain the folders and files that interest them. This workspace is unique to each user and is available to the user from any computer connected to the e-speak infrastructure. So unlike a web browser, e-speak allows you to take all of your file references with you when you change locations.

Users can share their cabinets with other users. For example, a provider could create a cabinet containing recipes and share this cabinet with other users. Someone using this service could create their own cabinet containing only those recipes that they like (using the browser application, the consumer could drag and drop the recipes from the providers cabinet to their own cabinet). If the provider updates a recipe (say, they change one of the ingredients), the consumer automatically sees this update the next time they access the recipe.

VFS was created to exercise the e-speak client library and core software. It is not intended to be a complete product. It attempts to exploit all the features of e-speak and, in doing so, provide a complete test application of the e-speak features.

SFS is an extension of VFS that allows the client to manage files based on attributes the client finds interesting such as the type of file, the creator of the file or the contents of the file. A client can define his/her own vocabulary and assign values to the attributes in that vocabulary and then structure the file system based on those attributes.

Overview of VFS concepts

The following graphic depicts the objects embodied by VFS and their relationship to each other.

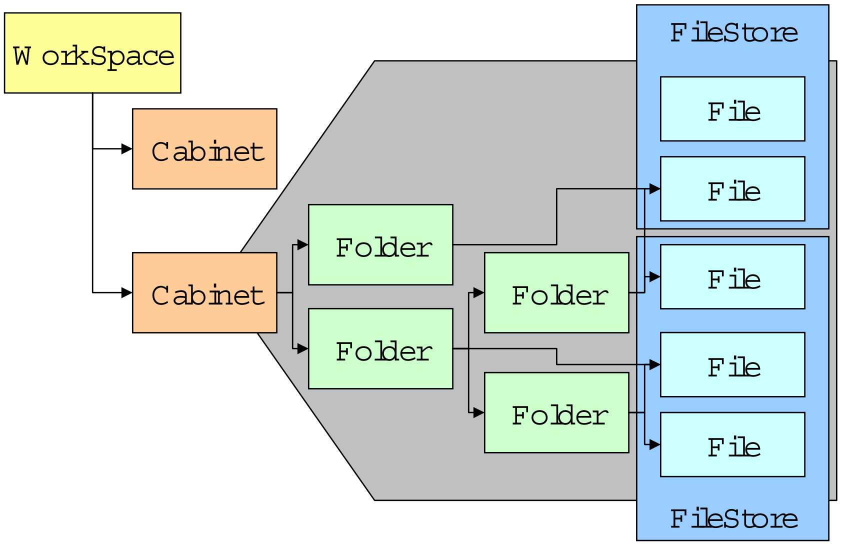


Figure 1 Relationship of VFS components

As can be seen in Figure 1, everything starts with the workspace. The workspace contains one or more cabinets. Each cabinet contains one or more folders. Each folder can contain other folders and/or files. A file can be referenced from more than one folder, or can be referenced from no folders. Each file is managed by one file store, but there is no relationship between file stores and folders or cabinets. A cabinet contains a reference to the default file store to simplify the creating of new files, but there is no policy that requires all files in a cabinet to be stored in the same file store.

Each of the components are described in more detail below:

Workspace

The workspace is associated with a user. It is analogous to the desktop metaphor of the Windows environment. The workspace is a named resource (typically with the user ID of the creator) and is discoverable to re-establish the previously saved environment. The workspace contains a list of Cabinets and provides the APIs necessary to manipulate these Cabinets. The workspace is also the anchor point for connecting to e-speak.

Workspaces are implemented using the ESFolder feature of e-speak.

Cabinet

The cabinet is used to collect folders and files. Much like an office file cabinet, it typically contains similar objects. Cabinets are also named and can be discovered. Cabinets can only contain folders and provide the APIs necessary to manipulate these folders.

Cabinets also contain a view (implemented using the Repository View feature of e-speak). This view is essentially a flattening of the folder and file hierarchy into a single list of files. A file can be contained in the view and not represented in the folder hierarchy of the cabinet. In this way, files can be added to the cabinet with no hierarchy required. (Note: This feature is currently not implemented)

Cabinets are implemented using the ESFolder and ESView features of e-speak.

Folder

The folder is used to hold other folders and files. Much like an office folder, it typically contains related objects. A hierarchy of folders simulates the hierarchy of files and directories typical in a file system. Folders can only contain other folders and files and provide the APIs necessary to manipulate these objects.

Folders are implemented using the ESFolder feature of e-speak.

File

The file represents a single, physical object located somewhere in e-speak. Files are created by a file store and can be referenced by a folder or cabinet view. The file provides simple file operation APIs necessary to fetch and store the contents of the file.

Files are implemented as remote resources.

File Store

The file store is an extension of the object store and is a remote service that provides the physical management of a group of objects (file objects in the case of VFS). The file store provides APIs to create and delete files.

A file store is implemented as a remote resource.

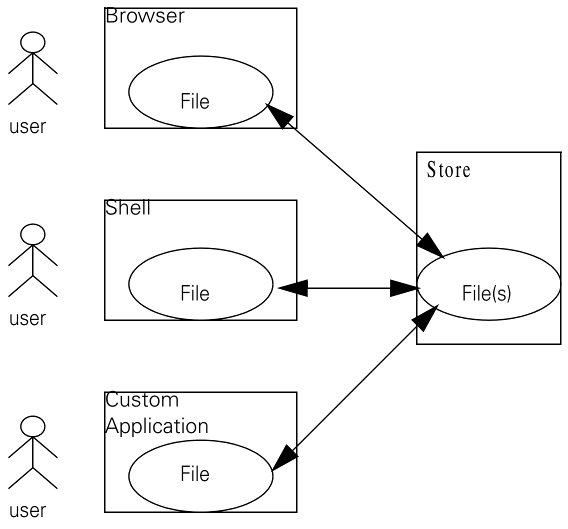


Figure 2 Use case view of VFS

Figure 2 shows the typical use cases for VFS. One or more users running either the Browser, Shell or another custom application using the VFS APIs interact with a File Store to store and fetch files.

Installation

The VFS source code was placed in the <installDir>/contrib/src/vfs directory when you installed e-speak.

If you want to run VFS, you can use the precompiled class files found in <install-Dir>/contrib/lib/vfs.

Using VFS

The VFS system is comprised of three major components. This section covers each of these components in more detail. The three components are:

- File Store
- Browser
- Shell

File Store

The file store is the service handler for the creation and deletion of files as well as the service handler for each of the files created by this file store.

When the file store starts, it first reads the properties file specified as the argument. The properties file specifies the e-speak connection parameters, the attributes of the file store and any other configuration items needed. It next connects to the e-speak core, initializes the directory where files are to be created, and registers itself with the core. It also advertises itself to the default community if requested.

The file store handles primarily two requests: createFile and deleteFile. For create-File, the file store creates a new file in the directory specified in the properties file, create an e-speak resource to represent this file, and return a VFSObject class to the caller. For deleteFile, the file store unregisters the file and deletes the physical file.

The file store process also handles requests for each of the files created by the file store. Again, there are only two basic requests: fetch and store. The fetch request returns the contents of the file as a byte array to the caller. The store request stores the byte array provided by the caller in the physical file. Each fetch and store is synchronized and buffered. Synchronization guarantees that a file cannot be read (fetched) until any active write (store) operation has completed and that a file cannot be written (stored) until all active read (fetch) operations have completed. Simultaneous reading (fetching) is allowed. Buffering guarantees that the physical limits of the core and the hardware does not cause a fetch or store I/O exception. The current buffer size is 64KBytes but can become a negotiable parameter in a future release.

Browser

The Browser is a Windows graphical client to the VFS system. To take advantage of automatic application launching and to be comfortable for Windows users, the Browser was written using the Microsoft extensions and therefor runs only on a Microsoft operating system (such as Windows NT). It uses the VFS APIs to communicate with the VFS file store and VFS files. It is modeled after the NT Explorer interface.

The application represents the workspace with each cabinet in a separate window within the application. The cabinet window is split into two primary panes. The left pane is the folder hierarchy and the right pane shows the contents of the selected folder. An optional third pane displays the cabinet's repository view contents.

The interaction is very similar to the NT explorer application. The user can click on the + sign next to a folder to expand that folder, select a folder, etc. In the right pane, if the user double-clicks on a folder it is expanded. A file can be dragged from one folder to another (this is a copy operation). Additional a folder can be dragged onto another folder (again, this is a copy).

If the user double clicks on a file, the browser copies the contents of the virtual file to the local hard drive (using the temporary directory) and ask Windows to launch the associated application. If the application changes the file, the browser prompts the user to save the changed contents back to the virtual file. If the user clicks OK, the temporary file is read and the data is stored back to the VFS file.

The context menus (right click on a pane or object) are used to access specific functions for the object in question. For example, right clicking on the left pane gives the user options to create new folders, import folders, etc. Right clicking on a folder gives the user options to export that folder, delete the folder, rename the folder, etc. Similar options exist for files.

When the user creates a cabinet, the user is asked to specify the default file store for that cabinet. This is used to specify the file store when importing files as well as the default for creating new files. This file store can be changed by clicking on file, cabinet, properties. It can also be overridden when creating a new file.

If the user renames a file in a folder, the user is only changing the local name of that file. The public name (the name used to search for the file or presented when a file is discovered) is not changed. To change the public name, the user must choose the rename context menu from the cabinet view pane (the third optional pane). The cabinet view pane can be displayed by selecting the view, cabinet contents menu.

If the browser is stopped and restarted (or started on another e-speak system), the user's workspace is discovered and all defined cabinets are opened. This demonstrates the ability for the VFS environment to follow the user wherever they are.

The user is able to view any cabinet within the workspace in a number of ways. First the user can view the contents of the selected cabinet. Alternatively the user can use the search console to find files in any cabinet by specifying the name of the file or something contained in that file. Finally, the user can create a semantic view of a cabinet by creating new constraints, modifying existing constraints or removing constraints from the semantic view. This allows the user to create a semantic view that includes all the files in the cabinet that, for example, were created after a certain date. The constraints that can be invoked are a function of the vocabulary used.

Shell

The shell package implements a simple command line interface (CLI) to interact with the VFS file store. The shell can be run with input scripts that are very similar to shell scripts in conventional shells. Since the purpose was mainly to enable automatic tests to be added to the test suite, the shell is not yet very user friendly. Most of the commands are still quite rudimentary and do not take all the options that are available in other shell scripting languages.

The shell behaves like a client of the file system. It looks for the workspace of the user who runs the shell and creates a new workspace if it cannot find the user's workspace. Once the user's workspace is set up, the shell allows the user to create and navigate an arbitrary hierarchy with the help of commands that are similar to commands that appear in standard UNIX shells.

Commands

The following is a list of commands that are accepted by the shell. We first provide a one-line description of these commands, and subsequently, explain the behavior of each command in more detail. Almost all the commands in shell are similar to commands with a similar name on UNIX systems.

cat	This command prints the contents of a file to the current output stream.
cd	This command changes the current working directory of the shell.
cp	This command copies files and sub-directories.
echo	This command streams the specified text to the current output stream.
exec	This command executes native binaries and shell scripts.
export	This command exports (writes out) a VFS folder (-d option) or a VFS file (-f option) onto the local disk.
find	This command searches prints out all of the files and directories in the specified directory.
import	This command imports a directory on the local file system as a folder in VFS (-d option) or imports a file on the local disk as a VFS file (-f option).
lock	This command explicitly locks a file for reading (-r option) or writing (-w option).
ls	This command lists the contents of the specified directory.
man	This command describes the use of the specified command.
mkdir	This command creates VFS directories.
mkfile	This command creates VFS files.
mv	This command moves files or directories.
pwd	This command prints out the present working directory.
reset	This command resets shell variables to their default value.
rm	This command removes files or recursively removes directories and their contents.

rmdir	This command removes empty.
set	This command sets shell variables such as standard input and standard output of the shell to the specified value.
share	This command reports or changes the share status of a cabinet.
show	This command shows workspaces, cabinets, view or file lock information.
unlock	This command explicitly release a files read lock (-r option) or write lock (-w option).
exit/quit	This command exits the shell.

Configuring and Running VFS

This section outlines the steps necessary to configure and run the VFS components.

FileStore

The VFS File Store accepts a single required parameter which specifies the file name of a property file. The property file specifies the properties for the e-speak environment and the configuration for the file store instance to be started. The e-speak properties are explained in <which document>. The file store properties are explained in Table 1

Table 1 File Store Properties

Property	Value Type	Explaination
PublicName	String	Specifies the public name for this file store. This is the name presented to the user and stored in the FileStore attribute of each file resource.
Reliability	Float	Specifies the percentage odds that the contents of a stored file will be retrieved successfully.
Availability	Float	Specifies the percentage odds that the file store service is operating when the client requests it.
StorageCost	Float	Specifies the cost per megabyte per month for data stored in the file store.
TotalCapacit y	Long	Specifies the total capacity in megabytes available to any one user.

Table 1 File Store Properties

Property	Value Type	Explaination
NumThread s	Integer	Specifies the number of service threads to start for handling file requests (fetch and store operations). The property effectively gates the amount of traffic the file store supports.
Root	String	The path to the disk location where the real file contents are to be stored.

A sample property file is shown in Figure 3.

```
; General e-speak properties
username: File Store User
Password: passwd
Hostname: localhost
Portnumber: 12345
Protocol: TCP
Sessionname: File Store Session
Community: Local Community
Eventcontrol: 1
Homefolder: File Store Home
Separator: /

; File Store Properties
PublicName: Production File Store
Reliability: 99.9
Availability: 99.9
StorageCost: 0.05
TotalCapacity: 10
NumThreads: 10
Root: D:/FileStore
```

Figure 3 Sample File Store Properties File

To start a file store use the following command:

```
cd <installDir>\contrib\vfs\config
run -i VFSFileStore.ini
```

This command assumes that the VFS and e-speak classes are defined in the system classpath variable. Property.file is the file name for the property file similar to the sample file that contains the properties for your file store.

Browser

The browser also accepts a properties file similar to that of the file store. The difference is that none of the file store properties are required but an optional property called WorkspaceName can be specified. If this property is not specified, the work space is named from the user.name property supplied by Java. The browser is started with the following command:

```
cd <installDir>\contrib\vfs\config
run -i VFSBrowser.ini
```

As with the File Store, this example assumes that the classpath is already set. This command displays a splash screen followed shortly by the browser application. Unlike the File Store, this command must run with the Microsoft Visual J++ environment because it uses the Microsoft extensions.

VFSShell

The shell accepts a property file similar to that of the browser. Like the browser, the property file can contain a property called WorkSpaceName. If this property is not specified the work space uses the name from the Java property user.name. The shell is started with the following command:

```
cd <installDir>\contrib\vfs\config
run -i VFSShell.ini
```

The two optional parameters (INFILE and OUTFILE) allow the user to specify a file name for the input and output, respectively, for the shell instead of using stdin and stdout. If the INFILE parameter is specified, the shell terminates when end-of-file is reached on the input file.

VFS Design

This section outlines the design decisions taken in the implementation of the VFS system using the e-speak technology. It also explains the programming model that was used to implement the various abstractions in the VFS system.

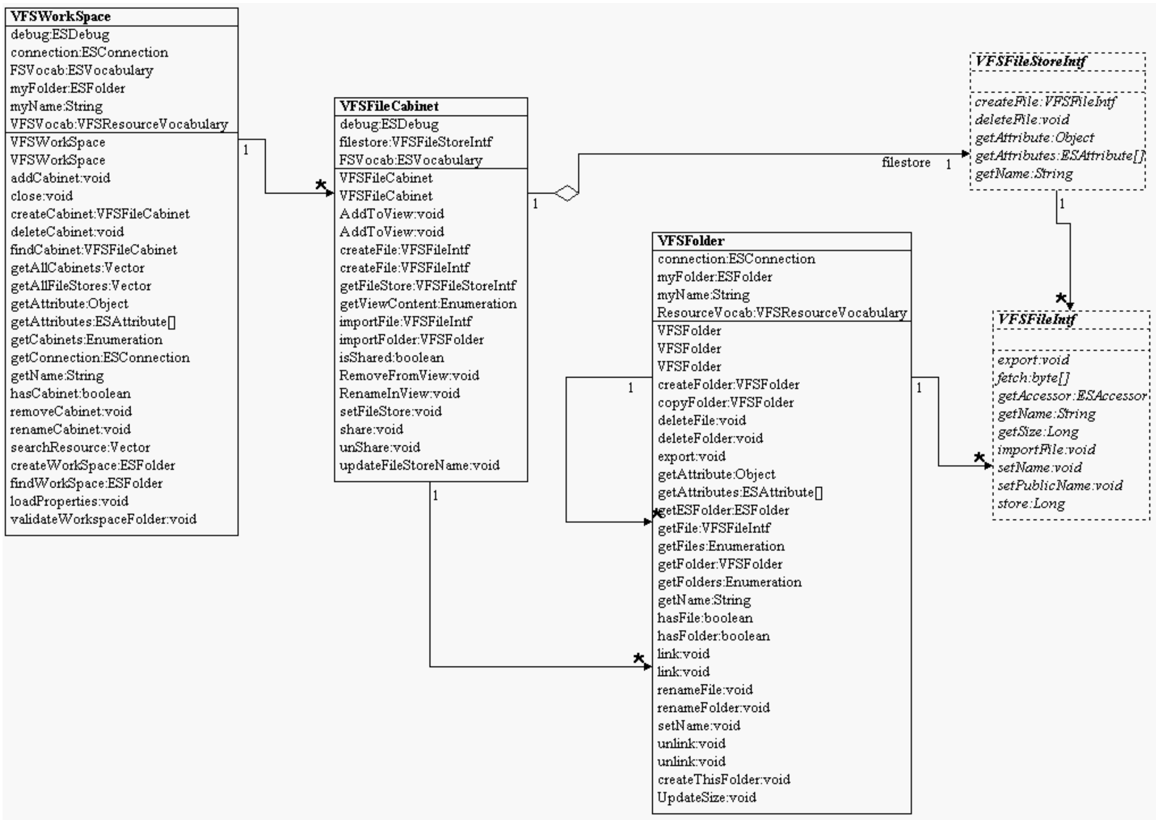


Figure 4 Key VFS class relationships

Figure 4 shows the relationships between the key VFS classes. We begin on the left with the VFSWorkSpace class. The work space can reference one or more cabinets. Each cabinet can reference one or more folders. Each folder can reference one or more other folders as well as one or more files. Each file is managed by a file store.

The cabinet has a reference to the default file store which it uses to create new files if a specific file store is not provided. However, each file reference by the cabinet/folder hierarchy can be managed by a different file store and therefore can be running on a completely different core. This demonstrates one of the key differences between VFS and a traditional file system.

Design of the Command Line Interface

This section outlines the design decisions taken in the implementation of the CLI for VFS. It explains some of the important limitations of the CLI when compared to other UNIX shells. It also explains how new commands can be added to the CLI.

The CLI shell is implemented with each command accepted by the CLI implemented as a separate class. The implementation of the shell is in class vfs.shell.VFSShell. Each command is implemented in the class vfs.shell.VFS<command>Command (source contrib/vfs/src/vfs/shell/VFS<command>Command.java).

The basic interpreter loop reads a command from the input stream and writes the resulting output to the current output stream. The current input stream could be “standard input”, i.e. the CLI console, or any file that has a sequence of commands for CLI. The CLI expects each command to be in a separate line and the parser uses white spaces as separators. Shell scripts can invoke other shell scripts. To make this possible, the shell maintains a stack of the relevant state for each script so that when a new script is to be executed, the state of the previous script gets pushed on top of a stack. This stack gets popped when the new script ends its execution so that the execution of the original script can continue. Currently, the scripts that the shell accepts are rather simple scripts. It does not yet support variable declarations within the script, nor does it have a convenient way to identify the arguments to the script within the script. These are limitations of the shell as a scripting language. The shell also does not support any form of partial matching or wildcards.

The shell is capable of executing executables from the local disk, however the CLI shell is not capable of automatically intercepting the output of the native executable to make it an entity in the VFS world. Nor is it yet capable of passing VFS files as arguments to native executables. For instance, it is not capable of invoking notepad on a VFS file and expects that each save of notepad result in the contents of the VFS be updated. What can be done, however is that a script can be written that exports a VFS file to a well known place, invokes notepad on it, and re-imports the file after the execution of notepad has completed. Similarly, if the native executable echo is executed, the shell is not capable of trapping the output of echo.

To add a new command to CLI, follow these steps.

1. Implement the desired functionality in class vfs.shell.VFS<command>Command. Follow the example of the other commands (e.g. vfs/shell/VFSlsCommand.java).

Design and implementation of the file and file store abstractions

The file resource is implemented as a remote resource in the VFS system. Users access the file abstractions via the class vfs.clientapi.VFSObject which in turn makes calls to methods in vfs.intf.VFSObjectIntf. The four classes that contain the interface code that implement the file abstractions are:

- vfs.clientapi.VFSObject: This interface implements the methods that are used directly by the user. The method hides things like buffer size restrictions and adds any “sugar” that is needed to make the object interface more user friendly.
- vfs.intf.VFSObjectIntf: This interface contains the signatures of the methods that are to be implemented by the filestore service. This file is auto generated by the ESIDL compiler based on the contents of vfs/intf/VFSObjectImpl.esidl.
- vfs.intf.VFSObjectStub: This is the client's view of the VFS object. It extends ESServiceStub and implements the stubs that provide the client with a network object model view of the service. This file is auto generated by the ESIDL compiler based on the contents of vfs/intf/VFSObjectImpl.esidl.
- vfs.server.filestore.VFSFileImpl: This class is an extension of vfs.server.ObjectImpl and contains the actual implementation of the file operations. This is the actual resource handler for the file resource.

Consider the fetch method excerpted from vfs/clientapi/VFSObject.java (the other methods have been deleted here for the sake of brevity):

```
public class VFSObject implements VFSObjectIntf {
   <other methods deleted..>
   /**
      * This method fetches the contents of the file into a bytearray.
      *
      * @return bytearray containing contents of file.
      * @exception ESInvocationException Someone serious is wrong communicating
      *             with the core.
    */

   public byte[] fetch()
   throws ESInvocationException {
      debug.enterProc(this, "fetch");
      // Lock this file for reading
      while (!file.testAndTry(READ_LOCK)) {}
      int count = (int)file.getSize();
      ByteArrayOutputStream out = new ByteArrayOutputStream(count);
      int offset = 0;
      int bufSize = MAX_BUFFER_SIZE;
      while (offset < count) {
         int whatsLeft = count - offset;
         if (whatsLeft < MAX_BUFFER_SIZE) {
            bufSize = whatsLeft;
         }
         out.write(file.fetchBuffer(offset, bufSize),
               offset, bufSize);
         offset += bufSize;
      }
      while (!file.release(READ_LOCK)) {}
      debug.exitProc(this, "fetch");
      return out.toByteArray();
   }
}
```

This method buffers the fetch by invoking the method fetchBuffer. The fetchBuffer method in vfs/intf/VFSObjectIntf.java, look as follows:

```
public interface VFSObjectIntf extends ESApplicationIntf {
   <other methods deleted..>
   public byte[]fetchBuffer(int offset, int size)
   throws ESInvocationException;
}
```

The corresponding methods in vfs/intf/VFSObjectStub.java, look as follows:

```
public class VFSObjectStub extends ESServiceStub
   implements VFSObjectIntf, ESSerializable{
   <other methods deleted..>
   public byte[] fetchBuffer(int arg0, int arg1)
   throws ESInvocationException {
      byte[] result = null;
      ParameterList params = new ParameterList();
      params.addObject( new java.lang.Integer(arg0), "int" );
      params.addObject( new java.lang.Integer(arg1), "int" );
      try {
            Object retObj = this.invokeSynchronous("vfs.intf.VFSObjectIntf",
               "fetchBuffer", params);
            result = ( byte[] ) retObj;
         }
         catch( ESInvocationException ex ) { throw ex; }
         catch( ESException ex) {
            throw new UnexpectedExceptionException(ex);
         }
         return result;
      }
   }
```

The method above constructs a parameter list and invokes the fetchBuffer method with the same signature on the service provider class using the invokeSynchronous call in the client library. In fact, all the methods in VFSObjectStub.java that implement the methods declared in VFSObjectIntf.java and implemented in VFSObject.java have a similar structure.

However, at the service provider end, the methods with the same signature have the actual implementations that do the fetch and store from the relevant sources for the bytes that represent the contents of the file. The current VFS system uses the private DATA of the file resource to store the name of the actual file on the disk that pertains to this particular VFS file. Note that using private DATA allows for a variety of possible implementations for the files.

For example, the bytes that make up the VFS file can be stored as a row in a database table. The private DATA then contains the information that can enable the file handler to determine where exactly the bytes that make up the contents of the file exist. For instance, this information can be name of database + name of table + row id of row that actually has the bytes. This private DATA of the file gets set when this file is registered as a resource. Here is the code from vfs/server/filestore/VFSFileImpl.java that implements this method:

```
public class VFSFileImpl extends ObjectImpl {
   /**
      * This method fetches a buffer of size <size> beginning at offset
      * <offset> from the file into a bytearray.
      *
      * @param offset int containing an offset into the file
      * @param size int containing the number of bytes to fetch
      * @return bytearray containing a portion of file.
      * @exception ESInvocationException Someone serious is wrong
      * communicating with the core.
      */

   public byte[] fetchBuffer(int offset, int size)
```

```
    throws ESInvocationException {
       debug.enterProc(this, "fetchBuffer");
       String fileName = getFileName();
       // Log this event
       log("Fetch File", fileName);
       try {
          File file = new File(fileName);
          int count = size;
          int whatsLeft = (int)file.length() - offset;
          if (whatsLeft < count) {
             count = whatsLeft;
          }
          byte [] fileBuffer = new byte[(int)count];
          FileInputStream in = new FileInputStream(file);
          in.read(fileBuffer, offset, (int)count);
          in.close();
          ESMap changes = new ESMap();
          changes.add(VFSResourceVocabulary.ACCESS_TIME, new
                Timestamp(System.currentTimeMillis()));
          updateAttributes(changes);
          return fileBuffer;
       } catch (IOException ioe) {
          debug.caughtException(this, ioe);
          debug.dumpStackTrace(this, ioe);
          OutofOrderRequestException e = new
             OutofOrderRequestException("VFSObjectImpl:fetchBuffer: " +
             ioe.toString());
          debug.throwingException(this, e);
          throw e;
       } finally {
          debug.exitProc(this, "fetchBuffer");
       }
    }
}
```

The client stub class can also cache some information, especially meta-data about the resource, but that depends on each implementation. In general, the client stub does not cache any meta-data at all.

The file store implementation has a structure that is virtually identical to the implementation of the file implementation. The four classes that implement the file store resource are:

- vfs.clientapi.VFSStore: This interface implements the methods that are used directly by the user. The method adds any “sugar” that is needed to make the file store interface more user friendly.

- vfs.intf.VFSStoreIntf: This interface contains the signatures of the methods that are to be implemented by the file store service. This file is auto generated by the ESIDL compiler based on the contents of vfs/intf/VFSStoreImpl.esidl.
- vfs.intf.VFSStoreStub: This is the client's view of the file store. It extends ESServiceStub and implements the stubs that provide the client with a network object model view of the service. This file is auto generated by the ESIDL compiler based on the contents of vfs/intf/VFSStoreImpl.esidl.
- vfs.server.filestore.VFSStoreImpl: This class is an extension of vfs.server.StoreImpl and contains the actual implementation of the file store operations. This is the actual resource handler for the file store resource.

The example above suggests that this is a common paradigm for implementing services in the e-speak system. The interface file declares the signatures of the methods that can be used by clients of the service. The client stub class implements these methods by calling the methods with the same signature on the service provider class. Any service that wants to provide the client a network-object model of the service can be structured in this manner.

Attributes for each object

Attributes are used by e-speak to describe an object. Attributes are defined within a vocabulary that describes the valid attributes and their types. VFS is implemented using two vocabularies, one for the file store and one for the rest of the objects.

File store

The file store vocabulary contains attributes that describe the performance of the file store and the capacity and cost of the file store.

- The **Resource Type** attribute is a String value that indicates this is a file store resource.

- The **Reliability** attribute is a floating-point number between 0 and 1 which describes the level of reliability that the file store is guaranteeing for files stored in this file store. Reliability indicates the chances of being able to read or write to the file at any given moment.
- The **Availability** attribute is a floating-point number between 0 and 1 which describes the level of availability that the file store is guaranteeing for files stored in this file store. Availability indicates the chances that the file contents are available if an error occur while processing that file.
- The **Capacity** attribute is a long value that describes the number of megabytes available to each user.
- The **Storage Cost** attribute is a floating-point number that describes the dollar cost per megabyte per month for storing files in the file store.
- The **Public Name** attribute is a String value that provides a friendly name for the file store.

Workspace, Cabinet, Folder & File

The resource vocabulary contains attributes that describe the nature of these objects.

- The **Resource Type** attribute is a String value that describes the type of resource. Possible values are:
  - **File** indicating a File resource,
  - **Folder** indicating a Folder resource,
  - **Cabinet** indicating a Cabinet resource, and
  - **Workspace** indicating a workspace resource.
- The **ResourceSubType** attribute is a String value that describes the type of resource within the parent resource type. For a File resource, the sub-type attribute describes what type of file it is (Word document, spreadsheet, etc.). For Folders and Cabinets it describes the predominate File sub-type this folder or cabinet contains. For workspaces, this attribute is not used.

- The **Date** & **Time** attributes are Timestamp values that record the last date and time the associated operation occurred. There are three date & time attributes: Creation, Last Access and Last Modified.
- The **Size** attribute is a long value that indicates the size of the object. For Files, this is the length of the file in bytes. For all other objects, this is the number of objects contained by that object (i.e., the number of files contained by the folder).
- The **FileStore** attribute is a String value that indicates the file store associated with the object. For Files, this indicates the file store that manages this file. For Cabinets, this indicates the default file store used when creating files in this cabinet.
- The **PublicName** attribute is a String value that provides a friendly name for the object. When a Cabinet, Folder or File is discovered and placed in a local container, the public name is used as the default local name for the object. The user can rename this object in their name space without affecting the public name.

Source code files and their purpose

The source for VFS consists of eight packages. They are:

1. vfs.browser - The browser GUI application classes
2. vfs.clientapi - The client API classes
3. vfs.infr - The interface classes
4. vfs.properties - The attribute filter classes
5. vfs.server - The server classes
6. vfs.server.filestore - The file store server classes (extensions of vfs.server classes)
7. vfs.shell - The shell application classes
8. vfs.util - Utility classes for all the above

vfs.browser package

The Browser application is a sample VFS client application using the Microsoft Windows Foundation Classes (WFC) to manage the GUI interface. The only knowledge the browser has of e-speak is the Attribute, Event and Exception classes.

AboutDialog.java

This class provides the verbiage for the “about” tab.

AttributeItem.java

This class is used to contain each resource represented in one of the list view panes. This includes the right hand pane, the search pane, the cabinet contents pane and the Select file store dialog. This class extends the default ListItem class to support storing the resource object associated with the item.

AttributeView.java

This class is used to manage the GUI for all the list view panes. It provides functions common to all list view windows, such as AddCabinet, etc., which add the associated resource type to the view. It provides the view context menu to change the view style (list, small icon, large icon, and report). It supports the item activate method for file resources which is used to launch the file. It provides the view-properties context menu to display the attribute dialog for any resource.

AttributeViewListener.java

This class is used to implement the event listener interface and listens to events for all the items displayed in the attribute view window (right hand pane). The AttributeView.java class subscribes to the events and this class updates the display when notified.

BrowseDialog.java

This class implements a utility function to display the directory structure on the host system and allow the user to select a directory.

BrowsePane.java

This class implements the bulk of the functionality of the VFS Browser application. It provides the Explorer-like interface for Cabinets. The user has the option to create, delete and copy Folders and Files within this cabinet and between cabinets. If the View, Cabinet Contents menu item is selected, a third pane appears at the bottom of the window. This pane contains a list of all the resources defined within this cabinet (i.e., the contents of the Repository View associated with the Cabinet).

Browser.java

This class implements the WFC container object to manage the Multiple Document Interface frame. There are a limited number of menu items that are implemented here, most of the application is implemented in BrowsePane.java and Search-Pane.java.

CabinetAttributesDialog.java

This class implements a dialog to display and edit the attributes for a Cabinet. The only attribute that can be changed is the default file store for new files created in this cabinet.

ChangeColumnsDialog.java

This class implements a dialog to change the columns in the context option.

ChooseVocabularyDialog.java

This class implements a dialog to allow the user to choose a vocabulary. The user is allowed to select the vocabulary from the list of know vocabularies.

ConstraintDialog.java

This class implements a dialog to add or edit constraints in the semantic view. The user is prompted for the vocabulary, constraint, boolean test and value for up to five constraints.

CreateVocabularyDialog.java

This class implements a dialog to allow the user to create a vocabulary. The user is prompted for the name and the attributes of the new vocabulary.

DirectoryNode.java

This class is used to contain each folder displayed in the left-hand pane of the main window. It extends the default TreeView class to support storing the type and resource object associated with the item.

DragData.java

This class is used to contain the data being dragged from one window to another. In this case, it is simply an object representing the resource being dragged.

FileInfo.java

This class is a set of helper methods to wrap the Windows APIs for obtaining information about a file. Methods include:

- getFileTypeEnum which returns the list of valid file types on the hosting system
- getIcon which returns the icon index for the requested file
- getLargeIconIndex & getSmallIconIndex which returns the icon index for the requested file
- getTypeName which returns the file type associated with the extension of the requested file

FileStoreListDialog.java

This class implements a dialog to display all the known file stores in the local repository. The user is allowed to select the file store from the list.

FolderWatcher.java

This class implements the event processing required to watch for changes to the folder contents. If a new file or folder is added to the currently displayed folder, this class updates the attribute view by adding new file or folder. Likewise, deletions are also updated.

InputDialog.java

This class implements a utility function to prompt the user for a input value.

LaunchFile.java

This class implements the launching of the associated application with the requested file. It fetches the file contents to a temporary location, determines which application is associated with the file and launches that application. It waits either for the application to terminate or for the file to be changed. If the file is changed, it prompts the user to save the changed contents back to the file store.

MessageBoxOnTop.java

This class implements a dialog similar to the Microsoft MessageBox command, except that this dialog is forced to be on top of all other windows. This was necessary since this dialog is displayed from a separate thread and the dialog could be lost behind other windows.

NewFileDialog.java

This class implements a dialog to specify the attributes for a new file. The user is prompted for the file name and the file store to store the file. The user is provided options for using the default file store or specifying the attributes of another file store to use.

ProgressDialog.java

This class implements a dialog to display the current progress during long operations. In particular, this dialog is used for importing folders.

PropertiesDialog.java

This class implements a dialog to display all the attributes set for a particular resource.

SFSBrowser.java

This class implements the semantic view window. It allows the user to create and name a view and to specify one to five constraints for searching the local repository. Any files that satisfy the query are placed into the view folder.

SFSColumn.java

This class is a helper method for managing the columns of the semantic view.

SFSNode.java

This class puts the name of the SFS node into the dialog.

SearchPane.java

This class implements the search console window. It mimics the NT Find window by allowing the user to specify one or more attributes to constrain the search of the local repository. Once one or more resources are found, they can be dragged to other parts of the application. A Cabinet can be dragged to the Container frame to open that Cabinet. A Folder or File can be dragged to an open Cabinet to link that folder or File to the Cabinet. In all cases, the drag operation is a copy operation.

SelectCabinetDialog.java

This class implements the dialog for choosing an existing cabinet. This dialog is displayed when the user selects the File, Open Cabinet menu item. The local repository is searched for all cabinets and the user is presented with the list.

SplashScreen.java

This class implements the splash screen that appears briefly when the application is started. Currently, it only displays a simple picture.

SysImageList.java

This class provides an extension to the Imaglist class provided by Microsoft. It is used for displaying the appropriate ICON image for files and folders.

TreeWatcher.java

This class is used to implement the event listener interface and listens to events for all the items displayed in the tree view window (left hand pane). The Tree-Watcher.java class subscribes to the events and this class updates the display when notified. The current implementation provides support for only some events. When the event system provides detailed information about the event this class can be enhanced to provide complete update.

vfs.clientapi package

The VFS API classes define the application-programming interface to communicate with a Virtual File System. None of these classes are runable, rather they are invoked by classes in the other sections.

The files are:

NoFileStoreFoundException.java

This class implements the only exception defined for VFS. This exception is thrown when a request is made to select a file store but no matching file store could be found. It extendsESServiceException to allow client applications to catch this exception along with other e-speak exceptions if they so choose.

SFSFolder.java

This class manages the semantic view folder. The SFSFolder contains a list of files that match the constraint set for this folder. If the folder is a sub-folder of another folder, the names in this folder are always a subset of the names in the parent folder.

VFSAttribute.java

This class provides an extension to the standard ESAttribute by also keeping track of the vocabulary the attribute is in.

VFSCabinet.java

This class implements the Cabinet construct. This class extends the VFSFolder.java class and therefore has all the functions of the Folder construct. Note, however, that while technically a Cabinet can contain Files, the intention is not to do so. Cabinets also define a Repository View to contain all the resources defined within the Cabinet.

Typically an application gains access to this class by calling one of the methods of VFSWorkSpace (e.g., createCabinet();).

In all cases, an application gains access to this class as a return value from VFSHelper.findFileStore();). The application should never create this class on it's own.

VFSFolder.java

This class implements the Folder construct. VFSFolder uses an ESFolder to implement the contents of the folder. Names are inserted into the ESFolder for each sub-folder and each file contained in the folder.

Typically, applications gain access to this class via one of the other classes methods (e.g, getFolders();).

VFSObject.java

This class is a value-added class that wraps the VFSObjectStub class. It provides higher-level functions to simplify the callers coding by providing the methods necessary to interact with a virtual file. VFSObjects are created by the VFSStore object and can be discovered via the VFSResourceVocabulary.

VFSResourceContract.java

This class implements the contract class for the file resource. This class implements theESContract class and provides the methods to interact with the contract.

VFSResourceVocabulary.java

This class contains a collection of read-only values to facilitate the naming of the various VFS attributes. Applications should use the strings defined here to specify attributes instead of hard-coding them.

VFSStore.java

This class is a value-added class that wraps the VFSStoreStub class. It provides higher-level functions to simplify the callers coding by providing the methods necessary to interact with a file store.

VFSStoreContract.java

This class implements the contract class for the File Store. This class implements theESContract class and provides the methods to interact with the contract.

VFSStoreDescription.java

This class extends the ESServiceDescription class and provides a type-safe implementation of the service description information.

VFSStoreElement.java

This class extends the ESServiceElement class and provides a type-safe implementation of the service element functionality.

VFSStoreFinder.java

This class extends the ESAbstractFinder class and provides find methods that are tailored to the VFS File Store environment.

VFSStoreVocabulary.java

This class contains a collection of read-only values to facilitate the naming of the various file store attributes. The constructor insures the e-speak vocabulary exists. Applications should use the strings defined here to specify attributes rather than hard-coding them.

VFSWorkSpace.java

This class implements the virtual workspace construct. VFSWorkSpace uses an ESFolder to implement the contents of the workspace. Names are inserted into the ESFolder for each Cabinet contained in the workspace.

This is the entry point for applications using the VFS system. The constructor searches the local repository for an existing workspace by this name and uses it if one is found. Otherwise, it creates a new empty workspace. The workspace creates and manages Cabinets. The workspace also supports the search function to locate VFS resources within the local repository.

vfs.intf package

VFSObjectIntf.java

This file is automatically generated by the ESIDL compiler. It defines the interface methods of the VFSObject construct. It is not called or accessed directly.

VFSObjectStub.java

This file is automatically generated by the ESIDL compiler. The class implements the virtual file construct. It is a front-end to the file construct, which is implemented by the server. The main functional methods are fetch and store. In most cases, an application gains access to this class as a return value to one of the other classes (e.g., Folder.getFiles();).

VFSStoreIntf.java

This file is automatically generated by the ESIDL compiler. The class defines the interface methods of the VFSStore construct. It is not called or accessed directly.

VFSStoreStub.java

This file is automatically generated by the ESIDL compile. The class implements the virtual file store construct. It is a front-end to the file store construct, which is implemented by the server. The main functional methods are createFile and deleteFile.

vfs.properties package

The properties classes implement the semantic file system attribute filtering functions. The properties services filter a new VFS object when it has been written to extract additional attributes from the contents of the object.

PropertiesContract.java

This file defines the constants used to describe the contract for a properties service.

PropertiesImpl.java

This file provides an abstract implementation for a properties service. All real property services must extend this class to reduce the development effort and take advantage of global changes.

PropertiesIntf.java

This file is automatically generated by the ESIDL compiler. It defines the interface methods of the Properties construct. It is not called or accessed directly.

PropertiesService.java

This file represents the office properties service to the e-speak core. It provides the management interfaces necessary to manage this service from a management environment. If supports registering, unregistering, starting, stopping, etc., the service via the management interface.

PropertiesStub.java

This file is automatically generated by the ESIDL compiler. The class implements the properties construct. It is a front-end to the properties construct, which is implemented by PropertiesServer.java..

PropertiesVocabulary.java

This class contains a collection of read-only values to facilitate the naming of the various properties attributes. The constructor insures the e-speak vocabulary exists. Applications should use the strings defined here to specify attributes rather than hard-coding them.

office/OfficePropertiesImpl.java

This file provides the implementation support for the Microsoft Office document filtering service.

office/OfficePropertiesService.java

This class implements the management interfaces for the Office Properties service environment.

office/OfficePropertiesVocabulary.java

This class contains a collection of read-only values to facilitate the naming of the MS Office properties attributes. The constructor insures the e-speak vocabulary exists. Applications should use the strings defined here to specify attributes rather than hard-coding them.

office/StartOfficeProperties.java

This class starts a instance of the Office Properties service. It handles files with the extension specified by the fileType property value.

vfs.server package

The server classes implement the Resource Handler functions for the external resources defined with VFS. This includes the file store and the File constructs.

ObjectImpl.java

This class implements the resource handler for the VFS Object construct.

StoreImpl.java

This class implements the resource handler for the VFS Store construct.

StoreService.java

This class represents the object service to the e-speak core. It provides the management interfaces necessary to manage this service from a management environment. If supports registering, unregistering, starting, stopping, etc., the service via the management interface.

filestore/StartFileStore.java

This class starts the file store service resource handler. It connects to the core, creates a ServiceContext and uses the management interface to register and start the service.

filestore/VFSFileImpl.java

This class extends ObjectImpl.java. It implements the resource handler methods specific to the VFS File construct. It is created by the file store and supports two simple file I/O operations: storeBuffer and fetchBuffer.

filestore/VFSFileStoreImpl.java

This class extends StoreImpl.java. It implements the file store resource handler methods specific to the VFS File construct. It handles all the file store calls from the client side interface. The two main functions are createObject and deleteObject.

filestore/VFSFileStoreService.java

This class extends StoreService.java. It provides the management interfaces specific to the File Store construct.

vfs.shell package

The shell package implements a sample VFS client application that supports a command line interface (CLI) to VFS. This is useful for creating test suites that can be automatically run to test VFS and the underlying e-speak software. The following classes make up the implementation of the CLI shell:

VFSShell.java

This file contains the implementation of the shell. The CLI shell gives the user a unix shell like environment for creating and browsing files in a directory structure. The shell is also capable of taking as input sequences of commands and redirecting output to any file in the local file system. The shell is also capable of importing and executing local executables in the foreground and background as in normal Unix shells.

VFSCommand.java

This file contains the abstract class that is overridden by all the VFS command classes.

VFS<command>Command.java

These files contain the implementation for each of the commands. For example, the implementation of “ls” is contained in VFSlsCommand.java.

VFSShellException.java

This file implements an exception that indicates an error was encountered processing a command.

VFSStack.java

This file contains a stack implementation that is used by the shell to keep track of the context of execution when embedded scripts are called. It is also used by VFSShell to navigate paths that are parts of commands.

2.0.1 VFSStackException.java

These are the two kinds of exceptions in the shell. The only externally visible exception is VFSShellException. VFSStackException is used internally in the shell to do error handling with stacks.

vfs.util package

VFSStrings.java

This class provides all the english message templates used by VFS.

Programmers Reference

For a detailed description of the VFS classes, refer to the web-based documentation at javadoc\overview-frame.html.

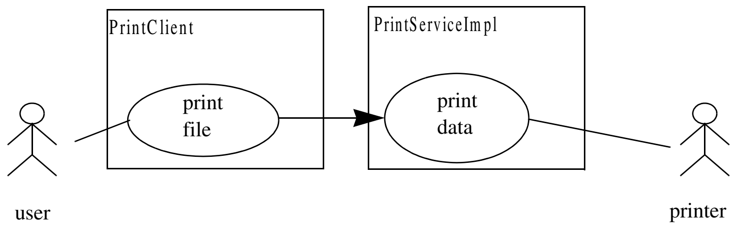
Chapter 3 Print Service

PrintServer uses the service advertisement mechanism and core software of the e-speak architecture. PrintServer is not intended to be a complete product.

Each print server knows how to print certain types of documents (e.g., .pdf, .ppt) in certain formats (e.g., color or duplex). Each print server has a speed and a quality rating. When the server starts, it registers its attributes with the e-speak core.

When a user selects a document and specifies the formats, speed and quality for printing the document, the print client requests a print service from the e-speak core. Based on the advertised services and the client criteria, e-speak gives the client a resource for performing the print job. The client passes the print job to the resource and, when the resource has finished with the job, it sends a message back to the client indicating completion.

A functional diagram of the PrintServer sample application running on the e-speak architecture.



Installation

The Print Client application uses the new swing classes for its user interface. Verify that your CLASSPATH environment includes the swing libraries by adding 'swing-all.jar' if you are using JDK1.1.x.

Run PrintServer and PrintClient

The following instructions describe how to run PrintServer and PrintClient. The print server is dependent on the perl helper program 'printit.pl' (specified in 'Print.ini') which on Windows-NT can print a variety of file types assuming that there are applications loaded that understand those file types. The provided sample program can run with PrintServer on PrintServer on UNIX with appropriate changes in the helper programs in 'Print.ini'.

Starting the Server

- cd into the Print Server config directory

  ```
  cd <installDir>\config
  ```
- Verify the 'Print.ini', and 'Printer.xml' settings and edit as needed. For more information about the configuration file format, see"“Setting up the environment” on page 52.
- Start the server

  ```
  ..\bin\espeak -i
  .\config\samples\printerserver\config\singlecore\ps.ini
  ```

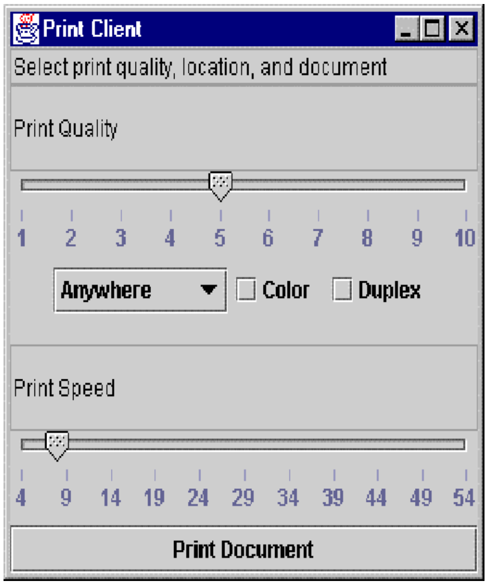
Starting the client

- cd into the Print Server config directory

  ```
  cd <installDir>\config
  ```
- Start the client

```
..\bin\espeak -i
.\config\samples\printerserver\config\singlecore\pc.ini
```

- The client appears on the screen.



- Select the parameters of the print job, select the file, and the client discovers the server that can service the request.

Using the print service across cores

A key-feature of e-speak is its ability to advertise a service across many cores such that the service is transparently available to remote machines The service provider must explicitly advertise the service.

The print server uses the advertising service so that it can be located from other cores sharing the same group server. Simply start one core (including the connection factory and ad service) with the server, and then start a second core (same requirements) with the client, and they find each other automatically.

Starting the server

- cd into the PrintServer config directory

  ```
  cd <installDir>\config
  ```

- start the server

  ```
  ..\bin\espeak -i ..\samples\printerserver\config\multicore\ps.ini
  ```

Starting the client

- cd into the PrintServer config directory

  ```
  cd <installDir>\config
  ```

- You can start the Core, Connection Factory and Advertising service to talk to the first core by

  ```
  ..\bin\espeak -i
  ..\samples\printerserver\config\multicore\PC-Core.ini
  hostname=<your hostname>
  ```

- Edit the property file used by the PrintClient and make changes to reflect the run location or the second core and the group server name. The property file is specified as an argument to the PrintClient in ‘PC.ini’.
- Start the PrintClient

  ```
     ..\bin\espeak -i
  .\config\samples\printerserver\config\singlecore\pc.ini
  ```

Configuring up the PrintServer environment

If you are not running e-speak using Sun JDK1.2 then you need to download the 'Swing' java extensions from Sun in order to use the print client. Swing is available from http://java.sun.com/products/jfc

To compile the PrintServer application, cd into the PrintServer directory and type 'compile' on Windows, or 'make' on UNIX This compiles the Java source and copies the .class files to the lib directory.

Print.ini file format

The Print.ini file configures what file types are supported, and what command to issue for each file type. The format of this file is shown below:

filetypes section

- The file extension and the associated command to print the file Variables %%temp_file%% and %%file_type%% are replaced by the PrintServer when the job is sent to the printing application

For example:

```
[filetypes]
ppt=perl printit.pl %%temp_file%% %%file_type%%
pdf=echo Could not print Acrobat file %%temp_file%% of type %%file_type%%
doc=perl printit.pl %%temp_file%% %%file_type%%
xls=perl printit.pl %%temp_file%% %%file_type%%
txt=notepad /p %%temp_file%%
ini=notepad /p %%temp_file%%
```

This example uses the printit.pl sample utility to print ppt, doc, and xls documents to the default system printer on the PrintServer. Notepad is used to print txt and ini files. If a pdf file is sent to the PrintServer, it is not printed, but the echo command sends a message back to the PrintClient.

Note that when using the sample printit.exe, the PrintServer system must have the associated applications, Word, Excel, PowerPoint, to print the associated document types.

Printer.xml file format

The Printer.xml file configures the way that the print server advertises itself to clients. You can specify for example the quality, speed, file types, and color capabilities of the printers that the server has access to.

For example, the following Printer.xml file describes a single HP LP5 printer capable of text output.

```
<?xml version="1.0"?>
<ESpeak version="E-Speak 1.0beta" operation="RegisterService"
xmlns="http://localhost/e:/Esxml/Schemas/espeak.xsd">
  <resource>
   <resourceDes xmlns="" name="HP Vectra PC">
      <!-- Specify PC Supplier Vocabulary -->
      <query xmlns="">
         <queryBlock xmlns="">
            <WHERE xmlns="">
               <!-- absence of query implies Base Vocabulary -->
               <condition xmlns="">
                  <IN xmlns="">
                     <pattern xmlns="">
                        <ResourceName xmlns="">printervocab</ResourceName>
                        <ResourceType xmlns="">Vocabulary</ResourceType>
                     </pattern>
                  </IN>
               </condition>
            </WHERE>
         </queryBlock>
      </query>
      <!-- Begin: attributes -->
      <attrSet xmlns="">
         <!-- End: Use PC supplier vocabulary -->
         <attr xmlns="" name="Manufacturer" required="true">
            <value xmlns="">HP</value>
         </attr>
         <attr xmlns="" name="Model" required="false">
            <value xmlns="">LP 5</value>
         </attr>
         <attr xmlns="" name="DPI" required="false">
            <value xmlns="">1200</value>
         </attr>
         <attr xmlns="" name="Speed" required="false">
            <value xmlns="">10</value>
         </attr>
         <attr xmlns="" name="Quality" required="false">
            <value xmlns="">10</value>
         </attr>
         <attr xmlns="" name="FileType" required="false">
            <value xmlns="">txt</value>
         </attr>
         <attr xmlns=”” name=”Location” required=”false”>
            <value xmlns=””>Lobby 49U</value>
         </attr>
         <!-- End: attributes -->
      </attrSet>
   </resourceDes>
  </resource>
</ESpeak>
```

Client-property-file format:

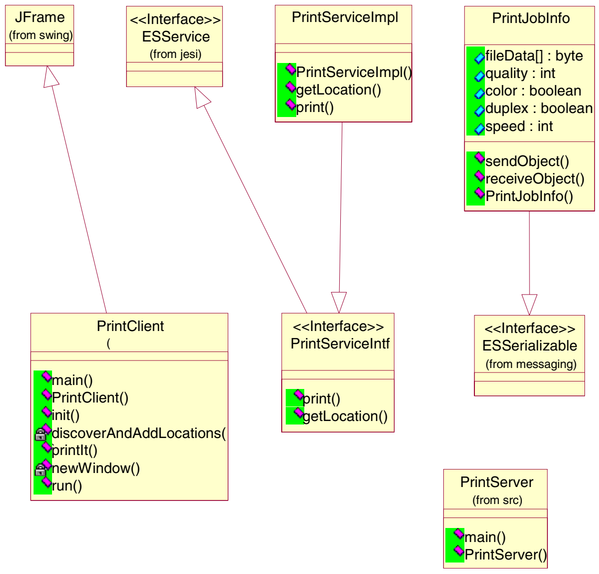
```
hostname=<hostname of e-speak core>
portnumber=<port of e-speak core>
community=<group server name>
```

An example client-property-file can be found in ‘<installDir>\samples\Print-Server\config\client.prop’:

```
portnumber=12346
community=printgrp
```

Design

This section details the design of the print example implementation. It also explains the programming model used to implement the various abstractions in the print example system.



PrintServer

The actual print server program.

PrintClient

A basic application that presents the user with a list of print job options, prompts the user for a file, discovers the PrintServer and processes the print job.

PrintJobInfo

Contains a single print job including the bytes required to represent the file to print.

PrintServiceIntf

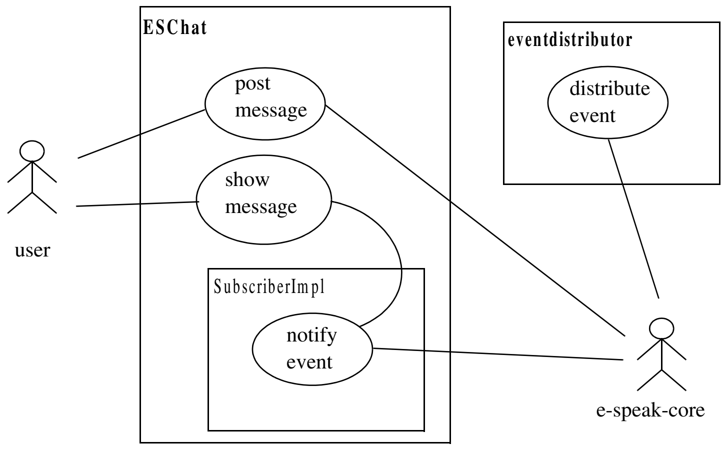
This is the print service interface which doubles as the skeleton.

PrintServiceImpl

The print service implementation.

Chapter 4 Chat Service

Chat was created to exercise the event mechanism and core software of the e-speak architecture. Chat is not intended to be a complete product.



How to run Chat

The following instructions describe how to run Chat.

Setting up the environment

The Chat example runs with JDK 1.1.x, JDK 1.2 or Microsoft's JVIEW The 'Swing' libraries must be installed on the system (they are bundled with JDK 1.2). Swing is available from http://java.sun.com/products/jfc/download.html

The CLASSPATH must be set to include the swingall.jar package, e-speak's JAR's and the local directory.

To compile the chat application, cd into the <installDir>\samples\ESChat directory and type Makefile This compiles the Java source and copies the .class files to the lib directory.

Starting the Server

- cd into the chat directory

  ```
  cd <installDir>\src\samples\ESChat\config\singlecore
  ```
- Start the event distributor

  ```
  run -i ESChatServer.ini
  ```

Starting the client

- cd into the chat directory

  ```
  cd <installDir>\src\samples\ESChat\config\singlecore
  ```
- Start the chat client

  ```
  run -i ESChatClient.ini
  ```
- The client appears on the screen.
- Click the Join button to join the chat group.
- Enter text into the input field, click the Post button and the message is sent to the server, which distributes it to the clients.

Configuration

The chat client '.pr' files use property files to set the location and port number of the e-speak core to connect to.

The format of this file is:

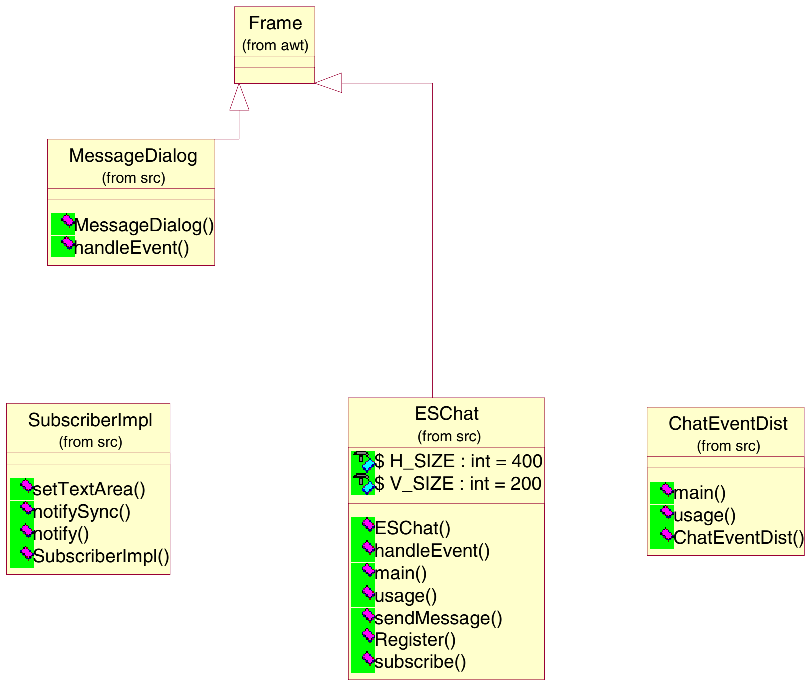
```
hostname=<hostname>
portnumber=<port number>
```

Other than the contents of the property files, there is no difference between the files in the singlecore directory and those in the multicore directory.

Design

This section outlines the design of the chat system implementation. It also explains the programming model used to implement the various abstractions in the chat system.

Source Code and Purpose



ESChat

This is the main class for the Chat application user interface. It instantiates an event publisher (for sending chat messages) and the event subscriber (for receiving chat messages.) It also presents the graphical user interface for receiving and sending messages.

SubscriberImpl

The SubscriberImpl class implements the ESListenerIntf so that messages can be received.

```
notify(
   Event e)
```

The notify method is activated when the subscriber receives a message sent by one of the chat clients. The message data is stored in the payload part of the event (extracted using getPayload()) which is then dynamically downcast from Object to Message.

```
notifySync(
   Event e)
```

This method receives synchronous event notifications but is not used in this example (the client doesn't send synchronous events).

MessageDialog

The message dialog class is used by the client to display any errors that happen during processing.

ChatEventDist

This class is an external event distributor which instantiates ESDistributor. It distributes events of type '*espeak.eschat*', an arbitrary string that was chosen to represent chat events.

Index